MASSACHUSETTS INSTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

VLSI Memo No. 88-468
August 1988

MESSAGE-DRIVEN PROCESSOR ARCHITECTURE

William Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Peter Nuth, Jerry Larivee, and Brian Totty

Abstract

The Message Driven Processor is a node of a large-scale multiprocessor being developed by the Concurrent VLSI Architecture Group. It is intended to support fine-grained, message passing, parallel computation. It contains several novel architectural features, such as a low-latency network interface, extensive type-checking hardware, and on-chip memory that can be used as an associative lookup table.

This document is a programmer's guide to the MDP. It describes the processor's register architecture, instruction set, and the data types supported by the processor. It also details the MDP's message sending and exception handling facilities.

Microsystems Research Center Room 39-321 | Massachusetts Institute of Technology | Cambridge Massachusetts 02139 | Telephone (617) 253-8138

Acknowledgements

This work was supported in part by the Defense Advance Research Projects Agency under contracts N00014-80-C-0622, N00014-87-K-0825 and N00014-85-K-0124, the National Science Foundation Presidential Young Investigators Award, General Electric Corporation and IBM Corporation.

Author Information

Dally, Chien, Fiske, Horwat, Keen, Nuth, and Larivee: Department of Electrical Engineering and Computer Science, Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139. Dally: Room NE43-417, (617) 253-6043; Chien: NE43-411, (617) 253-8572; Fiske, Horwat and Keen: Room NE43-416, (617) 253-8473; Nuth and Larivee: NE43-415, (617) 253-6048; Totty: Department of Electrical and Computer Engineering, University of Illinois, Urbana/Champaign, 1406 West Green Street, Urbana, IL 61801, (217) 333-2300.

MIT Concurrent VLSI Architecture Memo 14

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo No. 1069 August 18,1988

Message-Driven Processor Architecture[1] Version 11

William Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Peter Nuth, Jerry Larivee, and Brian Totty[2]

Abstract

The Message Driven Processor is a node of a large-scale multiprocessor being developed by the Concurrent VLSI Architecture Group. It is intended to support fine-grained, message passing, parallel computation. It contains several novel architectural features, such as a low-latency network interface, extensive type-checking hardware, and on-chip memory that can be used as an associative lookup table.

This document is a programmer's guide to the MDP. It describes the processor's register architecture, instruction set, and the data types supported by the processor. It also details the MDP's message sending and exception handling facilities.

Keywords: Processor Architecture, VLSI, Parallel Processing, Message Driven Processor, Fine Grain, Networks, Cache, Concurrent Smalltalk

[1]This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. The research described in this paper was supported in part by the Defense Advance Research Projects Agency of the Department of Defense under contracts N00014-80-C-0622 and N00014-85-K-0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation.

[2]Thanks also to Linda Chao, Soha Hassoun, Paul Song, and Scott Wills.

Table of Contents

Introduction

The Message-Driven Processor is a processing node for the J-Machine, a message-passing concurrent computer. The MDP is a standalone processor designed to provide support for fine-grained concurrent computation. Towards this goal the processor includes hardware for message queueing, low-latency message dispatching, and message sending. The same chip also contains a network interface and a router to allow the routing of messages throughout the network without any processor intervention.

The size of the MDP's register set is limited to minimize context-switching time. The memory is on the chip to improve performance and reduce the chip's pin count and the chip count of the concurrent computer. Having memory on-chip allows more flexibility in the use of memory than in designs with off-chip memory. For example, a portion of memory is designated as a two-way set-associative cache to be used by the XLATE instruction. Memory bandwidth is improved by providing row buffers that reduce the number of memory accesses required to fetch instructions and to enqueue messages, two operations which require frequent use of memory.

The MDP is also designed to efficiently support object-oriented programming. Every MDP word consists of 32 data bits and a 4 bit tag that classifies the word as an integer, boolean, address, instruction, pointer, or other data. In the MDP objects are described using a base address and a length, and all memory accesses are bounds checked. Memory addressing is normally done relative to the beginning or the head of an object. Absolute addressing is only done by the operating system. Having tags and no absolute references permits the use of garbage collection and transparent migration of objects to other MDP nodes on the network.

The MDP is almost completely message-driven. It is controlled by the messages arriving from the network that are automatically queued and processed. There are two priority levels to allow urgent messages to interrupt normal processing. There is also limited support for a background mode of execution when no messages are waiting in the queues.

This *Architecture* document is the assembly language programmer's manual for the MDP. It describes all of the MDP's features that are relevant to developing software for the processor. It does not describe the hardware of the chip in detail, nor does it explain the operating system used on the J-Machine.

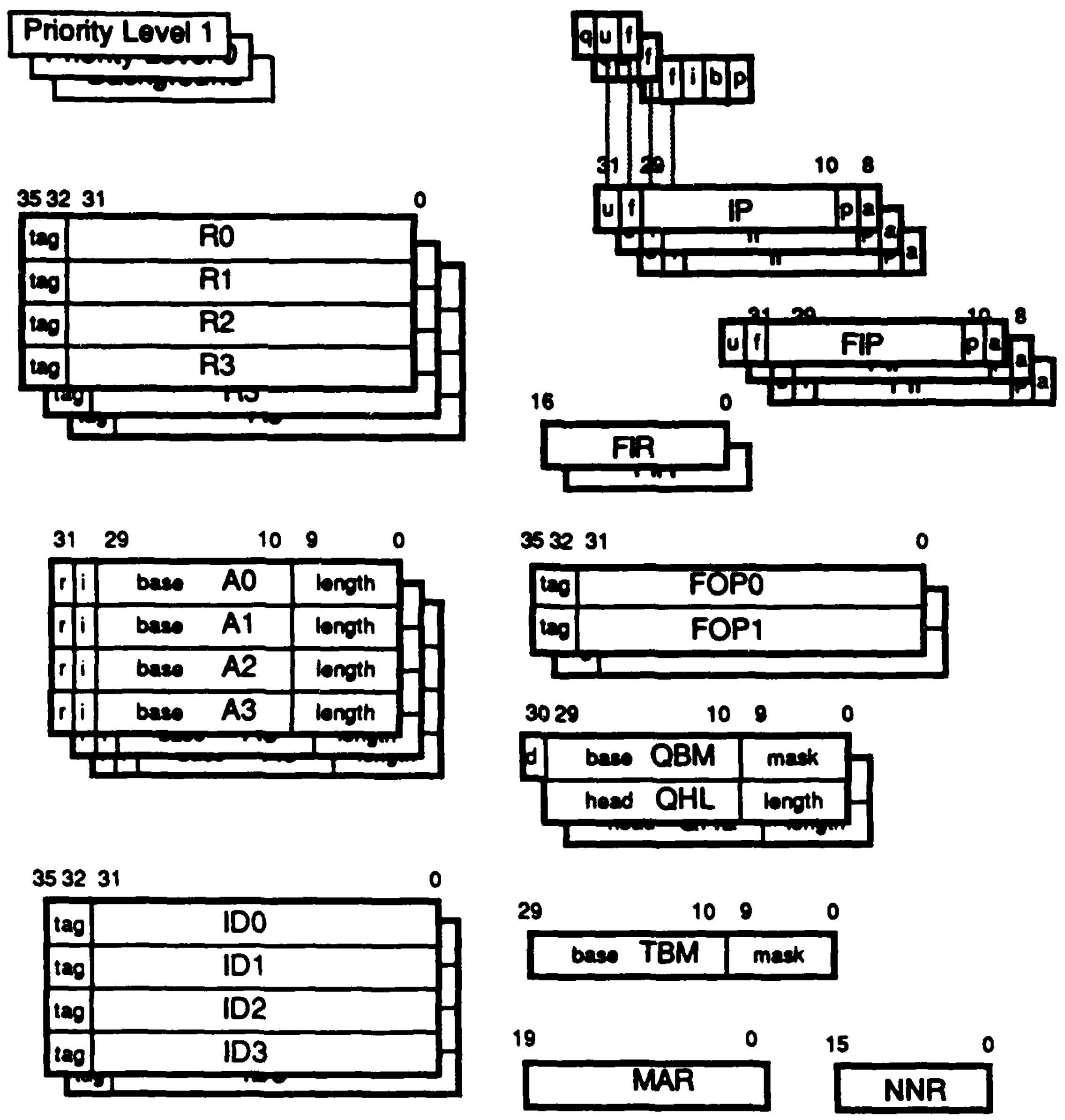
Changes since Version 10

The following changes have been made to the architecture since Version 10:

- More registers to support fault handling. Previously, we saved only the instruction register when a fault occurred. Now, we save: the instruction register (in FIR), the instruction pointer (in FIP) and the Op0 (in FOP0) and Op1 (in FOP1) operands.
- Inclusion of a set of registers for background mode. These include: 4 data registers, 4 address registers, an IP, and an FIP. There is no separate Q bit in the status register for background mode. There are separate U and F bits since they are part of the IP.
- The NNR (node number register) now has 3 fields instead of 2. This reflects the change in machine topology from 2D to 3D. Also, the NNR is no longer set to zero on a reset; it is left to software to initialize.
- Don't-care bits instead of 0's in unused positions in registers. For example, bits 0 to 7 of the IP register should now be considered to be don't-cares. No guarantees are made about their value.
- The WRITE instruction no longer supports the A addressing mode for Dst (op0).
- The RES (resume), MAX, and MIN instructions are no longer supported.
- The PUSH and POP instructions are no longer supported. Hence, stacks are no longer supported. LDIP and LDIPR are now available to accomplish reloading of the IP register when returning from a fault handler.
- The PURGE instruction is no longer supported. Instead, a NIL data value should be ENTERed for any key that you want to delete from the table.
- The PROBE instruction's operand format has been changed to resemble that of XLATE. That is, op0 specifies the Dst for the lookup, while op1 specifies the key for the search.
- The PROBE instruction now returns the data value associated with a key , rather than merely TRUE when a successful inquiry is made about a key's presence in the XLATE cache. If a key is not found in the cache, NIL is returned. The main difference between XLATE and PROBE is that XLATE faults if a key is not found or if a data value of NIL is associated with the key, whereas PROBE simply returns NIL. Note that PROBE now returns NIL rather than FALSE (as was the case for version 10) if the key is not found.
- The FFB (Find First Bit) instruction has been added. It is used to normalize floating point values.
- The branch instructions no longer support the A addressing mode for Src (op0). Src should be an integer, but the A registers cannot contain INT-tagged values.
- All 2-operand instructions that use the normal addressing mode for op0 have a 2 bit extension to the imm field in op0. Note that this applies to all 2-operand instructions, irrespective of which of op2 or op1 is unused. The 2 bit extension to imm comes from whichever of the fields op2 or op1 is unused. These 2 bits are the high order bits of the extended imm value. Also note that there are 2 types of imm that can be extended. We may use op0 to specify simply an immediate value, in which case this value is now 7 bits instead of 5. We may also use op0 to specify an offset into an object, in which case the offset is now a 6 bit immediate instead of 4.
- A similar extension to the op0 imm field is provided for 1-operand instructions that use the normal addressing mode. In such cases, the op2 field is always used for the 2 bit extension.
- A NOP instruction has been added. (opcode = 0)
- The ACCESS and RANGE faults have been eliminated, and the ILGADRMD fault has been merged into ILGINST.
- The size of Priority Switchable Memory has been increased from 16 words to 64 words.
- The fault vector table and call vector table have been separated, and two fault vector tables are now supported, one for each priority level. Also, with the removal of the RANGE fault, the call vector table is now of software definable length.
- An External interrupt has been added. The interrupt is handled as a fault.
- CATASTROPHE faults are now signaled if any fault occurs when the F bit is set. Also, the F bit now disables queue overflow and external interrupts when set.

- The I or the F bit disables queue overflow interrupts and external interrupts.
- A Memory Address Register, MAR, has been added for debugging purposes.
- An external memory interface has been added to support up to 1 MegaWord of DRAM.
- The SEND instructions set the I bit, and the SENDE instructions clear the I bit. Also, all SEND instructions use the Op2 field to encode which priority to send the message on.
- Tag checking on the special registers is only enforced for the Address and IP registers.
- The RTAG instruction now faults on CFUT's.

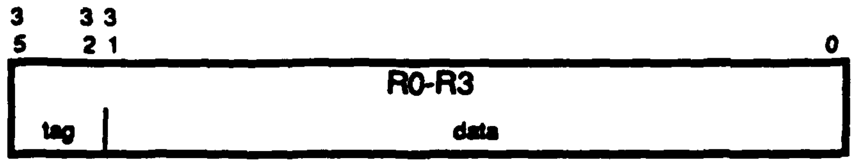
Processor State



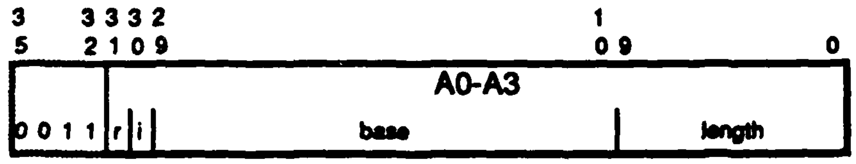
The processor state of the MDP is kept in a set of registers. There are two independent copies of most registers. One for each of the two priorities of the MDP, allowing easy priority switches while keeping the integrity of the registers. There is also a smaller, separate set of registers for background mode. There are no ID registers, no trap registers (FIR, FOP0, FOP1) except for an FIP register, and no queue registers (QBM, QHL) in the set of registers used in background mode. The registers are symbolically represented as follows:

• R0-R3	general-purpose data registers
• A0-A3	address registers
• ID0-ID3	ID registers
• SR	status register
• IP	instruction pointer register
• FIR	faulted instruction register
• FIP	faulted instruction pointer register
• FOP0	faulted OP0 register
• FOP1	faulted OP1 register
• QBM	queue base/limit register
• QHL	queue head/tail register
• TBM	translation base/mask register
• NNR	node number register
• MAR	memory address register

Register Descriptions



Four 36-bit general-purpose *data registers*, R0-R3, are capable of storing any word and tag. They are used for all data manipulation operations; as such, they are the most accessible registers in the programming model.



The *address registers*, A0-A3, are used for memory references, both data and instruction fetch. Each address register consists of a pair of integers and two bits. The integers represent the *base* and the *length* of an object in memory. The base points to the first memory location occupied by the object, while the *length specifies the length of the object.* The length field is used to support limit checking to insure that a reference lies within the bounds of the address register. A zero length specifies that no limit checking should be performed on accesses through the register, effectively making the object infinitely long.

Setting the invalid bit causes all memory references using the address register to fault INVADR. This fault is different from the one caused by referencing data of an object past its length limit.

The *relocatable bit* indicates that the address refers to an object that may be moved. *This bit allows a post-heap-compaction invalidation of only the* relocatable addresses, leaving the locked-down physical addresses intact.

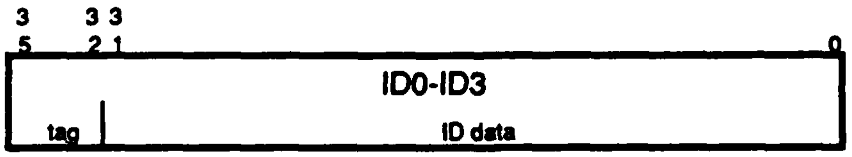
Address register 0 is used as the base register for instruction fetching; thus, it should point to the method currently executing. If, however, the A0 absolute bit in the IP is set, all reads, instruction fetches, and writes through register A0 ignore the value of register A0 and instead access absolute memory with an implicit base of 0 and unlimited length. This mode only affects memory accesses through register A0; the value of A0 can still be read and written normally.

Address register 3 is used as the pointer to the current message. When a message is dispatched the base and length of the message are written into A3. If the message has been copied into the heap, then A3 points into the heap; otherwise it points into the queue.

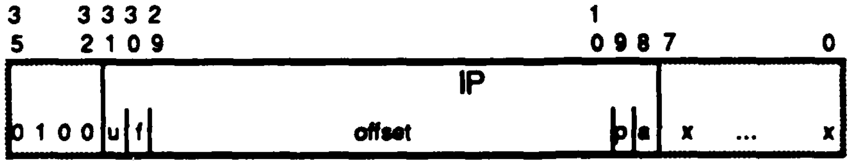
Address registers are read and written as ADDR-tagged values.

The following algorithm describes in detail the handling of a memory access through an address register:

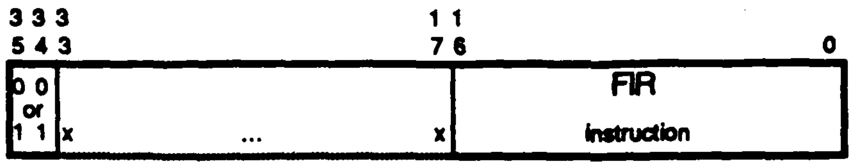
```
To access offset D from address register An:
    If n=0 and the A bit of IP is set (A0 absolute mode is active), access memory location D.
    Else
        If the I bit of An is set (An invalid), fault INVADR
        Else
            If length(An)≠0 and D≥length(An), fault LIMIT
            Else
                If the Q flag of SR is set and n=3
                    If D≥length(QHL), fault EARLY
                    Else access memory location base(QBM) ∨ ((base(A3)+ D) ∧ mask(QBM))
                Else access memory location base(An)+D
```



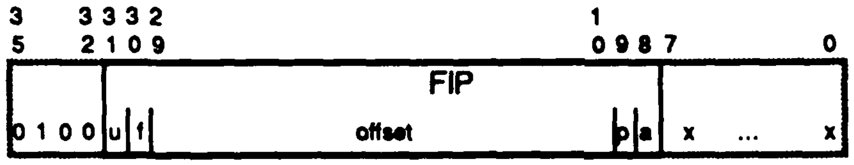
The four ID registers, ID0-ID3, exist to hold the IDs of relocatable objects in memory. In normal practice ID register n should hold the ID of the object pointed to by address register n. The ID is usually stored there by the XLATE instruction. When a fault occurs, the address register may be invalidated. Later, after the fault handler returns, an access through the address register causes an INVADR fault. The fault handler can then use the ID in ID register n to determine the new location of the object and the new value to be stored in the address register. The ID registers are shadowed by the address registers; this means that when XLATE'ing into an address register the corresponding ID register is written with the key that was XLATE'd.



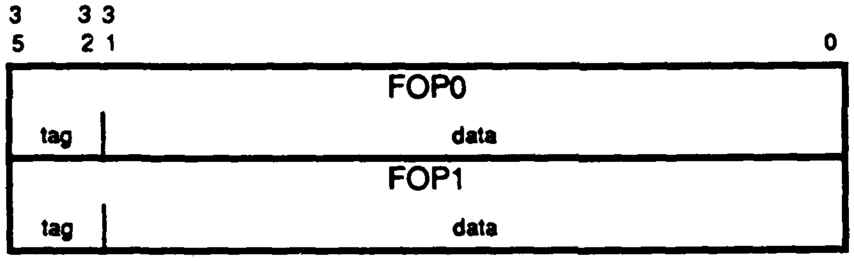
The *instruction pointer register*, IP, contains the offset within the object pointed to by A0 (or the absolute offset from the base of memory if A0 absolute mode is active) to the instruction following the instruction currently executing. Bit 9, the *phase bit*, specifies whether the low or the high instruction in the word pointed to by IP will be executed (high=0, low=1). That is, offset and phase together point to the next instruction to be executed. The *A0 absolute bit*, bit 8, when set, causes all memory references (read and write, data and instruction fetches) through register A0 to ignore the value of A0. This effectively allows absolute addressing of memory with an implicit base of 0 and an unbounded length. The value of A0 may still be read and written normally. Bit 31, the *unchecked mode bit*, is a copy of the unchecked mode flag in the status register. Changing it by changing the IP register changes it in the status register also and vice versa. Likewise, bit 30, the *fault bit*, is a copy of the fault flag in the status register.



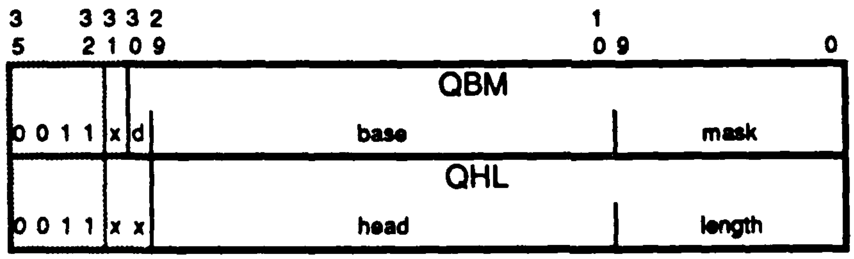
The *faulted instruction register*, FIR, contains the instruction that caused a fault while the fault handler is executing or NIL if the fault was not related to the execution of a specific instruction (i.e. an instruction fetch faulted, a bad message header arrived, a queue overflowed, etc.). It reads as either an INST-tagged value or NIL. Note that when FIR is non-NIL, the instruction is always given in the low 17 bits of FIR, even if it was fetched from the high 17 bits of a word in the execution stream.



The *faulted instruction pointer*, FIP, is loaded with the current IP when a fault occurs. Since the IP is pre-incremented, the FIP contains the IP to the instruction immediately following the faulting instruction.

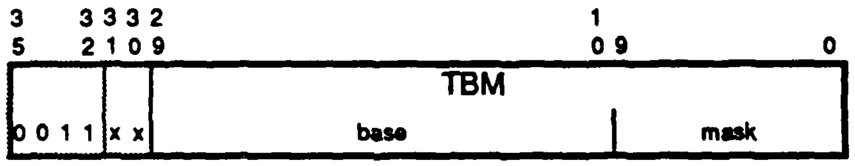


The *faulted operand registers*, FOP0 and FOP1, are loaded with the values of the Op0 and Op1 operands whenever an instruction-specific fault occurs. If a fault occurs that was not caused by a specific instruction, then the value written into these registers is indeterminate. If a faulting instruction has no Op0 or Op1 operand, then the value of FOP0 and/or FOP1 is indeterminate.

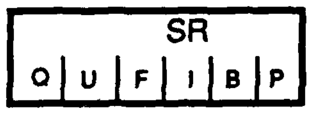


The *queue base/mask register*, QBM, contains the base and mask of the input message queue. The base is the first memory location used by the queue. The mask must be of the form 2^n-1, with $n \geq 2$. The size allocated to the queue is equal to the mask plus 1. There is one more restriction: $base \wedge mask = 0$ must hold. This effectively means that the base must be a multiple of the size of the queue, and this size must be a power of 2. These conditions allow queue access and wraparound to work by simply ANDing the offset within the queue with the mask and then ORing with the base. The *disable bit*, bit 30, should normally be zero. Setting it disables message reception at the priority level of the QBM register, which may cause messages to be backed up in the network. This should be done only under very special circumstances, such as when the queues are being moved. The QBM register is read and written as an ADDR-tagged value.

The *queue head/length register*, QHL, contains two fields, *head* and *length* that describe the current dynamic state of the queue. Head is an absolute pointer (i.e. relative to the beginning of memory, not the beginning of the queue) to the first word that contains valid data in the queue, while length contains the number of valid data words in the queue. The length is zero when the queue is empty, and 1 greater than the mask when the queue is full. QHL is read and written as an ADDR-tagged value.



The *translation base/mask register*, TBM, is used to specify the location of the two-way set-associative lookup table used by the XLATE and ENTER instructions. The format of the TBM register is similar to that of the QBM register. Again, base is the first memory location used by the table. The mask must be of the form 2^n-1, with $n \geq 2$. The number of words occupied by the table is equal to the mask plus 1. As in QBM, $base \wedge mask = 0$ must hold. TBM is read and written as an ADDR-tagged value.



The *status register* is a collection of flags that may be accessed individually using READR, WRITER, or the alias MOVE. The status register cannot be accessed as a unit. It contains these flags:

• P	current priority level (set: level 1, clear: level 0)
• B	background execution status (set: background, clear: normal (message))
• I	interrupt mask (set: no interrupts allowed, clear: interrupts allowed)
• F	fault (set: fault mode, clear: normal mode)
• U	unchecked mode (set: unchecked; clear:checked)
• Q	A3 queue wrap flag (set: A3 wraps around queue, clear: A3 normal)

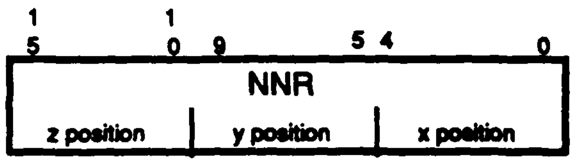
The *priority* and *background* flags specify the current priority level of execution. The highest level is priority 1, with the settings P=1, B=0. Below that is priority 0, with P=0, B=0. The lowest priority level is background, with B=1. When B=1, the P flag is ignored and the background register set is selected. An attempt to access a register that is not in the background set of registers produces undefined results.

The *interrupt mask flag*, in conjunction with the fault flag, determines whether the current process may be interrupted. Setting this flag disables all interrupts. Clearing this flag allows interrupts. There are three types of interrupts that may occur: priority switches, queue overflow interrupts, and external interrupts. Setting the interrupt mask flag disables all interrupts, clearing this flag allows priority switch interrupts and, if the fault bit is not set, also allows queue overflow and external interrupts. Allowing priority switches means that background processes may be interrupted by incoming messages at priority level 0 or 1 and level 0 processes may be interrupted by incoming messages at priority level 1. Queue overflow interrupts occur when the message queue for the current priority is full. External interrupts are explained in detail later.

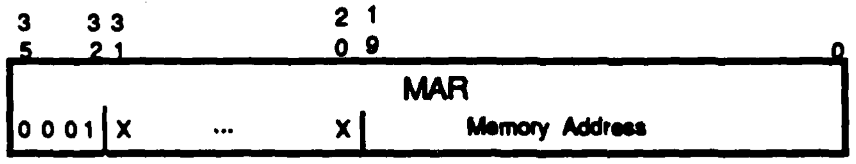
The *fault flag*, determines whether the occurrence of a fault would be lethal to the system and whether the process may be interrupted by a queue overflow or external interrupt. If a fault occurs while this flag is set, the processor faults CATASTROPHE, which should point to a special fault routine whose purpose is to clean up, if possible, and gracefully shut down the processor or the system. Queue overflow and external interrupts are disabled when this flag is set. This flag is loaded (with the new IP) when a fault occurs and cleared when the fault handler returns to the faulted program; it may, however, be altered by software as well. There is a copy of this flag in the IP register. Changing this flag changes it in the IP register and vice versa. There are three copies of the fault flag, one for each priority level and one for background mode. However, the background mode copy should never be needed in practice. No faults should occur in background mode. The flag exist simply because it is part of the IP format as well as being in the status register.

The *unchecked mode flag* determines whether TYPE, CFUT, FUT, TAG8, TAG9, TAGA, TAGB, and OVERFLOW faults are taken; when this flag is set, these faults are ignored, which allows more freedom in manipulation of data but provides less type checking. There is also a copy of this flag in the IP register. Changing this flag changes it in the IP register and vice versa. As with the F flag, there are three copies of the U flag, one for each priority level and one for background mode.

The *A3-Queue* bit, when set, causes A3 to "wrap around" the appropriate priority queue. This is included to allow A3 to act transparently as a pointer to a message, whether it is still in the queue, or copied into the heap. If the message is still in the queue, then setting the Q bit allows references through A3 to read the message sequentially, even if it wraps around the queue. If the message is copied into an object, then leaving the Q bit clear allows normal access of the message in the object. The Q bit is set on message dispatch, but it is left to the software to clear the Q bit when a message is copied into the heap. Either way, the access of the message pointed to by A3 looks like any other reference through an address register. Bounds checking is still performed using the length of A3 when A3 is referenced and the Q bit is set. Note that when the Q bit is set, the head of QHL should point to the same place as the base of A3 (since the start of the queue is also the start of the next message to be processed). There is a Q bit for each priority level, but no Q bit for background mode (because there is no queue for background mode).



The *node number register*, NNR, contains the network node number of this node. It consists of an X field, a Y field and a Z field indicating the position of the node in the 3D network grid. Its value identifies the processor on the network and is used for routing. The NNR should be initialized by software after a reset and left in that state. The NNR is read and written as an INT-tagged value.



The *Memory Address Register*, MAR, is provided for debugging purposes and should therefore be of little use to the applications programmer. This register contains the value of the most recent memory address generated by an op0 read or write; this is an absolute value from the base of memory. Note that this register is only written by op0 memory references and *not* instruction fetches or any other implicit memory references. The MAR is read only and cannot be written using the WRITER instruction.

Data Types

The following data types may be used in a word:

35–32	31	30	29–10	9–0	
0 0 0 0	value (0=NIL)				SYM
0 0 0 1	two's complement value				INT
0 0 1 0	0		...	0 b	BOOL
0 0 1 1	r	i	base	length	ADDR
0 1 0 0	u	f	offset	p a x ... x	IP
0 1 0 1	u	f	offset	length	MSG
0 1 1 0	user-defined				CFUT
0 1 1 1	user-defined				FUT
1 0 0 0	user-defined				TAG8
1 0 0 1	user-defined				TAG9
1 0 1 0	user-defined				TAGA
1 0 1 1	user-defined				TAGB
1 1	0 0 first instruction			second instruction	INST0
1 1	0 1 first instruction			second instruction	INST1
1 1	1 0 first instruction			second instruction	INST2
1 1	1 1 first instruction			second instruction	INST3

- *SYM* contains an atomic symbol. EQUAL and NEQUAL are allowed on SYMbols. If the data portion of a symbol contains all zeroes, the word takes on the value of NIL.
- *INT* contains a two's complement integer between -2^{31} and $2^{31}-1$, inclusive. All arithmetic, logical, and comparison operations are allowed on INTs.
- *BOOL* contains a boolean value, which is either true (b=1) or false (b=0). All logical, and comparison operations are allowed on BOOLs. For purposes of the comparisons, false is considered as less than true.
- *ADDR* contains a base/length pair that may be loaded into either one of the address registers or QBM, QHL, or TBM. The uses of bits 30 and 31 vary among these registers.
- *IP* contains a value appropriate for loading into the IP. See the IP register section for a description of the fields.
- *MSG* is the header of a message. It is similar to an IP except that it has no phase bit or absolute bit and the low order 10 bits contain the length of the message (including the MSG word).
- *CFUT* contains a context future. Almost all operations fault on context futures. They are not meant to be MOVEable. CFUTs are used as placeholders for values to be computed in parallel by other processes; an attempt to read a CFUT before its value is available will fault, and the operating system suspends the current process until the value is available.
- *FUT* is a standard future. FUTs may be moved, and their tags may be read and written, but they may not participate in any primitive operations such as addition or checking for equality. As with CFUTs, an attempt to use a FUT in a primitive operation causes a fault, and the operating system will have to provide the appropriate value for the FUT.

- *TAG8* through *TAGB* are tags for software-defined words. They cause faults on all primitive operations except EQ, NEQ, BNIL, and BNNIL.
- *INST0* through *INST3* are tags for instructions. The two instructions in a word occupy a total of 34 bits, so two tag bits are also used to encode them.

Memory

The prototype MDP contains 4096 words of RAM; there are 4096 words reserved for ROM, although not all of this reserved ROM space is actually used. If the MDP has external memory available, it is placed above the ROM. Future MDPs may have more memory and different address maps, so user programs should not rely on absolute memory locations other than the fault vectors.

Certain memory locations have special purposes assigned to them by the hardware. These are outlined in the table below.

From	To	
$00000	$0001F	Priority Switchable Memory 0
$00020	$0003F	Priority Switchable Memory 1
$00040	$0005F	Priority 0 Fault Vectors
$00060	$0007F	Priority 1 Fault Vectors
$00080	$00FFF	Uncommitted RAM
$01000	$01FFF	ROM
$02000	$FFFFF	External Memory address space

Within the uncommitted internal RAM, the operating system usually allocates the first few hundred words to the call vector table, the message queues, and the XLATE cache and leaves the rest of RAM for user programs. The call vector table length is operating system definable, but its base must be location $00080.

The External Memory Controller for the MDP supports dynamic memory refreshing and error checking / correction (ECC). The memory signals a DRAMERR fault when a double-bit error occurs; single-bit errors are corrected. Access to the Refresh Timer Counter, RTC, and the Error Counter, ERC, is provided through memory locations $2000 and $2001 respectively. The RTC is a 7 bit register. Writing the RTC sets the interval between refresh operations. Reading the RTC returns the current count. The Error Counter is an 8 bit register that is incremented every time a single bit error is detected. Only the bottom seven bits of the ERC are used as a counter; the 8th bit disables ECC when set.

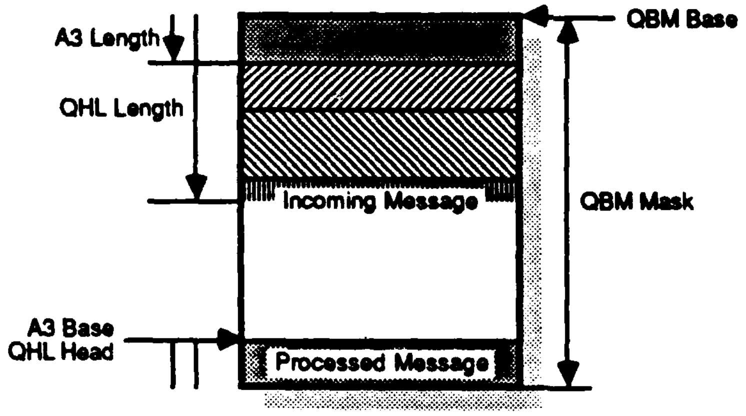
Priority-Switchable Memory

In order to allow each priority level to have 32 private temporaries, the first 64 words of memory are decoded specially. When accessing one of these 64 words, the current state of the P flag is XORed with bit 5 of the address; hence, referencing location 1 accesses physical location 1 when running in priority level 0 (P flag clear) or location 33 when running in priority level 1 (P flag set). This scheme lets the operating system and user programs use memory locations 0 through 31 as temporaries private to the current priority level. The other priority level's temporaries can be accessed as locations 32 through 63.

Network Interface

Message Queues

Incoming messages are queued in *message queues* before being dispatched and processed. There are two message queues, one for each priority level. Each message queue is defined by two registers—QBM, the queue base/mask register, and QHL, the queue head/length register. The queue base/mask register defines the absolute position and length of the queue in memory. In order to simplify the hardware, the length must be a power of 2, and the queue must start at an address that is a multiple of the length. The queue head/length register specifies which portion of the queue contains messages that have been queued but not processed yet (including the message not yet dequeued by SUSPEND). To avoid having to copy memory, the queue wraps around; if a twenty-word message has arrived and only eight words are left until the end of the queue, the first eight words of the message are stored until the end of the queue, and the next twelve are stored at the beginning. The queue head/length register contains the head and length of the queue instead of the head and tail to simplify the bounds-checking hardware involved in checking user program references to the queue. Below is a diagram of a queue with one message being processed, two more waiting, and a third one arriving.



Due to the presence of row buffers in the hardware, messages are always stored at multiples of four words in memory, sometimes causing there to be one, two, or three words of wasted space between messages in the queue. This alignment is transparent to the software; the length and head in QHL are automatically aligned to multiples of four words by the hardware. The length field of the message header specifies the exact length of the message.

When messages are dispatched, the A3 register is written with the base field from the QHL and the length field from the bottom 10 bits of the message header. The Q bit in the status register allows accesses to messages that are "wrapped around," such as the twenty-word message in the example above.

A message may interrupt lower priority processes and be dispatched as soon as the first queue row buffer is written into the queue; the processor does not wait until the entire message is present before dispatching it. Read accesses to words through A3 with the Q bit set are also checked against the length of the current message and the length of the queue; if the latter test fails, an EARLY fault is generated to indicate an access to data in the message that has not yet arrived. Writes through A3 are never checked for EARLY faults. Note that if the check against length of the current message fails, a LIMIT fault is generated instead. The EARLY fault is necessary because the length of the current message may be longer than the current length of the queue. When a message comes in, the header tells what the length of the complete message is; this is the current message length. The length of the queue indicates how much of the message has actually arrived.

Message Reception

There are two stages in processing of messages: queueing and execution. In general, incoming messages from the network are first queued in the priority 0 or 1 queue. When a message begins arriving in a queue, execution begins. If the message starts executing and references an item that is not yet in the queue, an EARLY fault occurs. There are a few places where delays could occur in the above procedure. These are outlined below.

- If the D bit of a QBM is set, the corresponding queue is disabled. Messages are not allowed into the queue until that bit is cleared. This may cause backups in the network.
- If a queue is full, the effect is the same as in the above situation. If the processor is executing at the same level of priority as the message and the F and I flags are clear, a fault is generated to warn the processor about the condition.
- The I flag in the status register prevents messages from interrupting lower priority processes when it is set. They may, however, be queued.
- An arriving message may interrupt a process running at a lower priority level but not one running at the same priority level. That is, priority level 1 messages may interrupt level 0 message handlers and background processes, while priority level 0 messages may only interrupt background processes.

When the processor begins executing a message, the B flag is cleared, P is set to the priority at which the message arrived on the network, and the IP offset is loaded from the first word of the message, which must be tagged MSG; if it isn't, a MSG fault is taken. The F and U flags are all loaded from the message header. A3 is set up to point to the message in the queue; the Q flag of the SR is always set. The A0 Absolute bit and Phase bit of the IP are set to 1 and 0 respectively.

SUSPEND

The SUSPEND instruction terminates the processing of the message. First it flushes one message from the proper input queue. Then, if another message (of either priority) is ready, it is executed as described in the **Message Reception** section. Otherwise, the IP is fetched from the background IP and execution resumes with the next instruction of background code. A SUSPEND executed in background mode produces indeterminate results.

Note that every message arrival corresponds to exactly one SUSPEND. This SUSPEND terminates the processing of the message and also flushes the message. Therefore, every MDP routine that gets executed by a message must terminate with a SUSPEND at some point.

Message Transmission

The SEND, SEND2, SENDE, and SEND2E instructions are used to send messages. The first word sent specifies the absolute node number of the destination node (i.e. the destination node's NNR value) in the low 16 bits. The SEND instruction uses the current node's NNR and the destination node number to find the relative offsets in the X, Y, and Z dimensions that the network controllers use in routing the messages through the network. The tag of the first word is currently ignored, although it is recommended that the tag be INT. The op2 field of each SEND instruction determines the priority at which the message is to be sent over the network: 0 means priority level 0 and 1 means level 1. The priority of the message is independent of the priority of the process that is sending it.

The initial routing word is followed by a number of words which the network delivers verbatim to the destination node. The network does not examine the contents of these words. The message is terminated by a SENDE or SEND2E instruction, which sends the last one or two words, and tells the network to actually transmit the message. The first word that arrives at the destination node (the second word actually sent, since the routing word is only used by the network and doesn't arrive at the destination node) must be tagged MSG. It contains the length of that message including the message header but not including the routing word preceding it. It also contains the initial value of the IP at which execution is supposed to start. The destination node faults MSG if this word is not tagged MSG.

The total time between the first SEND and the SENDE should be as short as possible to avoid blocking the network. To accomplish this the SEND and SEND2 instructions set the I flag and the SENDE and SEND2E instructions clear the I flag, thus disabling interrupts during message transmission. For the same reason, faults should be avoided while sending.

Exceptions

Reset

When the processor is reset, the status register flags are set as follows: Q=Q'=0, U=U'=1, F=F'=0, I=1, B=1, P=0. The A bit in the IP and D bits in both QBM registers are set. The background IP offset is set to the first location in ROM. The program that gets executed (starting at the first location in ROM) on a reset should set up the queues, NNR, and at least some of the fault vectors and then clear the I flag and the D bits in the QBM registers to allow message reception.

Fault Processing

When a fault occurs, the instruction that caused the fault is saved in the FIR register, the current IP (which points one instruction beyond the faulting instruction) is saved in the FIP register, and the values of the Op0 and Op1 operands (if any) are saved in the FOP0 and FOP1 registers; the IP is then fetched from the memory location whose address is equal to the fault number plus the base of the fault vector table of the current priority (when in Background mode the fault vector table for whichever priority is selected by the Priority flag is used). If the F bit is set and a fault occurs then the IP is loaded from the CATASTROPHE fault vector. The U, A, and F bits of the IP that gets loaded may change the processor state. U determines if this priority is in unchecked mode, A determines if A0 absolute mode is in effect, and the F bit determines whether the fault is non-reentrant and interruptible.

System Calls

A system call (via the CALL instruction) mimics some of the behavior of a fault to provide convenient access to system routines. When a CALL occurs, the base of the system CALL vector table is added to the CALL operand, and the contents of this location are fetched, yielding a call handler IP. The current IP (which points to the next instruction) is saved in the current priority's FIP register. Execution then begins by loading the call handler IP (which sets the F, A, and U bits in the status register to the values in the call handler IP).

Interrupts

There are three types of interrupts supported on the MDP: priority switches, queue overflow interrupts, and external interrupts. Priority switches may occur at any time, provided that the I flag is clear; queue overflow and external interrupts may only occur when both the I and F flags are clear. Priority switches should be the most common interrupts; these occur when a message arrives in the queue of a priority higher than the current priority. Thus, priority 1 messages can interrupt priority 0 or background mode, and priority 0 messages *can interrupt background mode. The handler for a priority switch is the interrupting message itself.*

Queue overflow interrupts are signalled when the last empty word of the queue is written, but may cause an interrupt only when running at the same priority as the queue which overflowed. In other words, if the priority 0 queue overflows and a priority 1 process is currently running then the handler for the queue overflow must wait until all pending priority 1 processes have suspended before it can start execution. Likewise, if the priority 0 queue overflows and a background mode process is currently running and either the F or I flag is set then the handler must wait until both flags are cleared before execution can begin. When a queue overflow interrupt is taken, a fault is signalled and the IP is loaded from the QUEUE fault vector.

External interrupts are similar to queue overflow interrupts except that whenever the I and F flags are clear and an external interrupt is signalled, a fault is signalled at the current priority and the IP is loaded from the INTERRUPT fault vector. The interrupt is handled as a process of the same priority as the priority which it interrupted. An external interrupt is signalled by an external interrupt pin on the MDP package.

Interrupts may occur only between instructions. After an interrupt the FIP points to the next instruction of the interrupted sequence.

The following faults are defined:

Name	Fault Number	Description
CATASTROPHE	$0	Double fault,bad vector, or other catastrophe.
INTERRUPT	$1	Interrupt pin has gone active.
QUEUE	$2	Message queue about to overflow.
SEND	$3	Send buffer full.
ILGINST	$4	Illegal instruction.
DRAMERR	$5	Double bit error in the external RAM.
INVADR	$6	Attempt to access data through address register with I bit set.
LIMIT	$7	Attempt to access object data past limit.
EARLY	$8	Attempt to access data in message queue before it arrived.
MSG	$9	Bad message header.
XLATE	$A	XLATE missed.
OVERFLOW	$B	Integer arithmetic overflow.
CFUT	$C	Attempted operation on a word tagged CFUT.
FUT	$D	Attempted operation on a word tagged FUT.
TAG8	$E	Attempted operation on a word tagged TAG8.
TAG9	$F	Attempted operation on a word tagged TAG9.
TAGA	$10	Attempted operation on a word tagged TAGA.
TAGB	$11	Attempted operation on a word tagged TAGB.
TYPE	$12	An operand or a combination of operands with a bad tag type used in an instruction.
	$13-1F	Reserved for future faults.

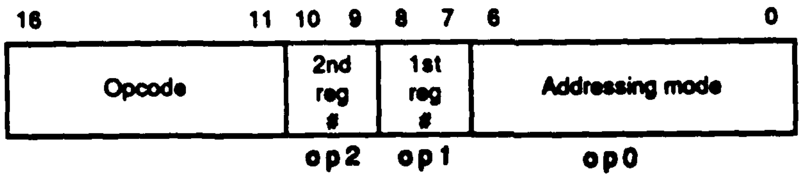
Note: If multiple faults occur simultaneously the fault vector chosen is the one that has the highest precedence. Each fault is assigned a precedence by its fault number; lower fault numbers correspond to higher precedence.

Instruction Encoding

The program executed by the MDP consists of instructions and constants. A constant is any word not tagged INST0 through INST3 that is encountered in the instruction stream. When a constant word is encountered, that word is loaded into R0 and execution proceeds with the next word.

Every instruction is 17 bits long. Two 17-bit instructions are packed into a word. Since a word has only 32 data bits, two tag bits are also used to specify the instructions. The instruction in the high part of the word is executed first, followed by the instruction in the low part of the word. As a matter of convention, if only one instruction is present in a word, it should be placed in the high part, and the low part of the word set to all zeros.

The format of an instruction is as follows:



The *opcode* field specifies one of 64 possible instructions. The other fields specify three operands; instructions that don't require three operands may ignore some of the operand fields. Operands 1 and 2 must be data registers; their numbers (0 through 3) are encoded in the *1st reg #* and *2nd reg #* fields. Operand 2, if used, is always the destination of an operation and operand 1, if used, is always a source.

In the case of 1-operand and 2-operand instructions that use op0 in the normal addressing mode, one of op2 or op1 is used to provide a 2 bit extension to an imm value specified in op0 (if an imm value is specified in op0). In the case of 2-operand instructions, the 2 bit extension is found in whichever of op2 or op1 is not used. The 2 bit extension is always in the op2 field for 1-operand instructions.

Operand 0 can be used as a source or a destination in an instruction. It can hold two possible encodings. A normal instruction has op0 address mode encodings as follows:

Normal Addressing Mode (bits 6 … 0)	Syntax	Addressing Mode
0 0 0 0 0 | Rn	`Rn`	Data register Rn
0 0 0 0 1 | An	`An`	Address register An
0 0 0 1 | 0 0 0	`NIL`	Immediate constant NIL (SYM:0)
0 0 0 1 | 0 0 1	`FALSE`	Immediate constant FALSE (BOOL:0)
0 0 0 1 | 0 1 0	`TRUE`	Immediate constant TRUE (BOOL:1)
0 0 0 1 | 0 1 1	`$80000000`	Immediate constant INT:$80000000
0 0 0 1 | 1 0 0	`$FF`	Immediate constant INT:$000000FF
0 0 0 1 | 1 0 1	`$3FF`	Immediate constant INT:$000003FF
0 0 0 1 | 1 1 0	`$FFFF`	Immediate constant INT:$0000FFFF
0 0 0 1 | 1 1 1	`$FFFFF`	Immediate constant INT:$000FFFFF
0 0 1 | Rx | An	`[Rx,An]`	Offset Rx in object An
0 1 | imm	`imm`	Immediate imm (signed)
1 | imm | An	`[imm,An]`	Offset imm (unsigned) in object An

The immediate constants are eight immediate values outside the range `INT:-16..INT:15`. They are provided for convenience and code density improvement. The $FF and $FFFF constants are useful for masking bytes and words, while the $3FF and $FFFFF constants may be used for masking lengths and addresses.

The imm field is extended by 2 bits for all 2-operand operations that use this normal addressing mode for op0. These extra 2 bits are obtained from either the op2 field or the op1 field (whichever one happens to be unused). The 2 bits serve as the high order bits of the extended imm value. If simply an immediate value is being specified by op0, then this value is now 7 bits instead of 5. In the case of an offset into an object, the offset is now a 6 bit immediate value instead of only 4. This extension allows much longer branch distances.

The register-oriented op0 mode is used instead of normal op0 mode by the READR, WRITER, and LDIPR instructions. The register-oriented op0 mode encodings are as follows:

6						0	Syntax	Addressing Mode
Register-Oriented Addressing Mode								
B	P	0	0	0	Rn		Rn	Data register Rn
B	P	0	0	1	An		An	Address register An
-	P	0	1	0	IDn		IDn	ID register IDn
B	P	0	1	1	0	0	FIP	Trapped Instruction pointer
-	P	0	1	1	0	1	FIR	Trapped Instruction register
-	P	0	1	1	1	0	FOP0	Trapped OP0 register
-	P	0	1	1	1	1	FOP1	Trapped OP1 register
-	P	1	0	0	0	0	QBM	Queue Base/Mask register
-	P	1	0	0	0	1	QHL	Queue Head/Length register
B	P	1	0	0	1	0	IP	Instruction Pointer register
-	-	1	0	0	1	1	TBM	Translation Base/Mask register
-	-	1	0	1	0	0	NNR	Node Number register
-	-	1	0	1	0	1	MAR	Memory Address Bus register
-	-	1	0	1	1	0		Unused (ILGINST fault)
-	-	1	0	1	1	1		Unused (ILGINST fault)
-	-	1	1	0	0	0	P	Priority Level flag
-	-	1	1	0	0	1	B	Background Execution flag
-	-	1	1	0	1	0	I	Interrupt flag
B	P	1	1	0	1	1	F	Fault flag
B	P	1	1	1	0	0	U	Unchecked flag
-	P	1	1	1	0	1	Q	A3 Queue flag
-	-	1	1	1	1	0		Unused (ILGINST fault)
-	-	1	1	1	1	1		Unused (ILGINST fault)

B represents the use of the Background register set or one of the two priority register sets. The B bit is XORed with the Background Flag and a register set chosen according to the result; 1 indicates the background registers, while 0 indicates the register set chosen by the P bit relative to the present priority.The assembler syntax for specifying a register belonging to the background is the register name followed by a "B".

P represents the priority of the register being accessed, and is relative to the current priority. 0 indicates the current priority, while 1 indicates the other priority. The assembler syntax for specifying a register belonging to the other priority is the register name followed by a backquote (`).

Certain registers are typed—their values always read as a given type, but attempts to write values of a different type do not fault. The address and IP registers however are checked on writes and writing a value of any value other type than that specified does fault TYPE, CFUT, FUT, TAG8, TAG9, TAGA, or TAGB except in unchecked mode, depending on the value that is attempted to be written. Below is a table of the types of the registers.

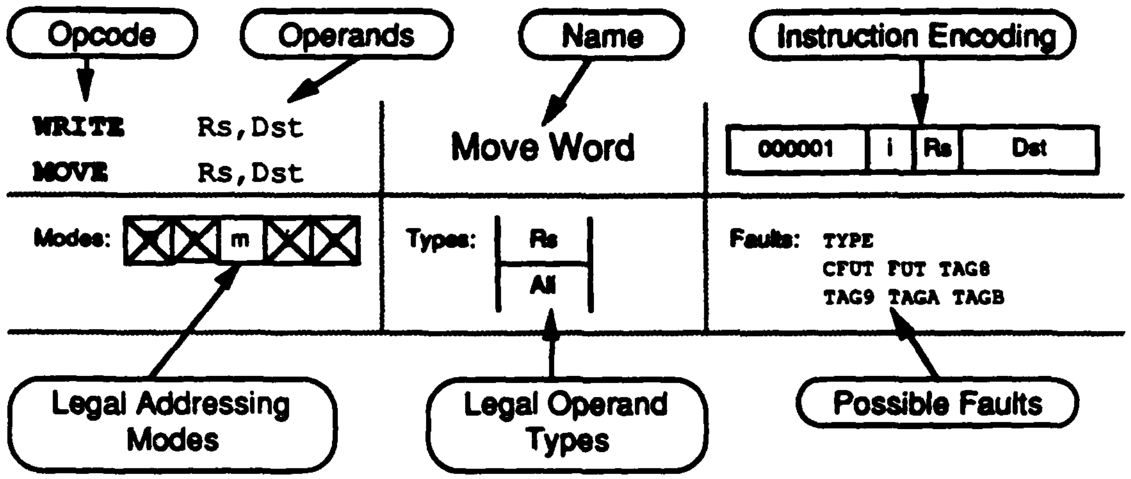
Register	**Type**
Rn	Any
An	ADDR
IDn	Any
QBM	ADDR
QHL	ADDR
IP	IP
FIR	Any
FIP	IP
FOP0	Any
FOP1	Any
TBM	ADDR
NNR	INT
MAR	INT
P	BOOL
B	BOOL
I	BOOL
F	BOOL
U	BOOL
Q	BOOL

Instruction Set Summary

Mnemonic	Operands	Name	Op	Modes	Types
General Movement and Type Instructions					
READ	Src,Rd	Move Word	$01	R,A,m,i,c	All but CFUT
WRITE	Rs,Dst	Move Word	$02	m	All
READR	Src,Rd	Read Register	$03	Register	All but CFUT
WRITER	Rs,Dst	Write Register	$04	Register	All
RTAG	Src,Rd	Read Tag	$05	R,A,m,i,c	All but CFUT
WTAG	Rs,Src,Rd	Write Tag	$06	R,A,m,i,c	All,Int
LDIP	Src	Load IP	$07	R,A,m,i,c	Ip
LDIPR	Src	Load IP from Register	$08	Register	Ip
CHECK	Rs,Src,Rd	Check Tag	$09	R,A,m,i,c	All,Int
Arithmetic and Logic Instructions					
CARRY	Rs,Src,Rd	Carry from Add	$0A	R,A,m,i,c	Int,Int
ADD	Rs,Src,Rd	Add	$0B	R,A,m,i,c	Int,Int
SUB	Rs,Src,Rd	Subtract	$0C	R,A,m,i,c	Int,Int
MULH	Rs,Src,Rd	Multiply High	$0E	R,A,m,i,c	Int,Int
MUL	Rs,Src,Rd	Multiply	$0F	R,A,m,i,c	Int,Int
ASH	Rs,Src,Rd	Arithmetic Shift	$10	R,A,m,i,c	Int,Int
LSH	Rs,Src,Rd	Logical Shift	$11	R,A,m,i,c	Int,Int
ROT	Rs,Src,Rd	Rotate	$12	R,A,m,i,c	Int,Int
AND	Rs,Src,Rd	And	$18	R,A,m,i,c	Int,Int or Bool,Bool
OR	Rs,Src,Rd	Or	$19	R,A,m,i,c	Int,Int or Bool,Bool
XOR	Rs,Src,Rd	Xor	$1A	R,A,m,i,c	Int,Int or Bool,Bool
FFB	Src,Rd	Find First Bit	$1B	R,A,m,i,c	Int
NOT	Src,Rd	Not	$1C	R,A,m,i,c	Int or Bool
NEG	Src,Rd	Negate	$1D	R,A,m,i,c	Int
LT	Rs,Src,Rd	Less Than	$20	R,A,m,i,c	Int,Int or Bool,Bool
LE	Rs,Src,Rd	Less Than or Equal	$21	R,A,m,i,c	Int,Int or Bool,Bool
GE	Rs,Src,Rd	Greater Than or Equal	$22	R,A,m,i,c	Int,Int or Bool,Bool
GT	Rs,Src,Rd	Greater Than	$23	R,A,m,i,c	Int,Int or Bool,Bool
EQUAL	Rs,Src,Rd	Equal	$24	R,A,m,i,c	Int,Int or Bool,Bool or Sym,Sym
NEQUAL	Rs,Src,Rd	Not Equal	$25	R,A,m,i,c	Int,Int or Bool,Bool or Sym,Sym
EQ	Rs,Src,Rd	Pointer Equal	$26	R,A,m,i,c	All but CFut or Fut
NEQ	Rs,Src,Rd	Pointer not Equal	$27	R,A,m,i,c	All but CFut or Fut
Network Instructions					
SEND	Src,P	Send	$34	R,A,m,i,c	All but CFut
SENDE	Src,P	Send and End	$35	R,A,m,i,c	All but CFut
SEND2	Src,Rs,P	Send 2	$36	R,A,m,i,c	All but CFut
SEND2E	Src,Rs,P	Send 2 and End	$37	R,A,m,i,c	All but CFut
Associative Lookup Table Instructions					
XLATE	Rs,Dst,C	Associative Lookup	$28	R,A	All but CFut
ENTER	Src,Rs	Associative Enter	$29	R	All but CFut,All but CFut
PROBE	Rs,Dst	Probe Associative Cache	$2D	R	All but CFut
Special Instructions					
NOP		NOP	$00		
INVAL		Invalidate	$2A		
SUSPEND		Suspend	$30		
CALL	Src	System Call	$31	R,A,m,i	Int
Branches					
BR	Src	Branch	$38	R,i	Int
BNIL	Rs,Src	Branch if NIL	$3A	R,i	All but CFut,Int
BNNIL	Rs,Src	Branch if Non-NIL	$3B	R,i	All but CFut,Int
BF	Rs,Src	Branch if False	$3C	R,i	Bool,Int
BT	Rs,Src	Branch if True	$3D	R,i	Bool,Int
BZ	Rs,Src	Branch if Zero	$3E	R,i	Int,Int
BNZ	Rs,Src	Branch if NonZero	$3F	R,i	Int,Int

Instruction Set

Below is a table of the instructions available on the MDP listed in numerical opcode order. The instructions are specified as follows:



The Legal Addressing Modes field specifies which addressing modes are legal with this instruction. Any illegal addressing modes are crossed out. *R* specifies data registers, *A* address registers, *m* memory (either [Rx,An] or [imm,An]), *i* immediate (a signed imm value), and *c* constant (one of the 8 immediate constants). If the register-oriented mode is used instead, the Modes field contains a single box with the words *Register Mode* in it.

The Legal Operand Types field specifies which combinations of types of operands are legal. Each row in the table indicates a legal combination of types. Some instructions have more than one combination of legal types. For these instructions a TYPE fault occurs if both types are legal but their combination is not. For example, the AND instruction faults TYPE if one of its operands is a BOOL and the other one is INT, even though it does accept two BOOLs or two INTs. Illegal types cause either a TYPE fault or CFUT, FUT, or one of the TAG faults. When two different words with bad types are used as arguments, the fault corresponding to one of them is signalled; it is not specified which has precedence.

The Possible Faults field specifies the faults that are possible with this instruction. The common faults (CATASTROPHE, ILGINST, ACCESS, EARLY, LIMIT, INVADR, and MSG) are not listed, as they may occur for almost all instructions and behave in the same way for all instructions.

NOP	NOP	000000 | 00 | 00 | 0000000

READ Src,Rd **MOVE** Src,Rd	Move Word	000001 | Rd | i | Src
Modes: R | A | m | i | c	Types: Src: All but CFUT	Faults: CFUT

Rd ← Src

WRITE Rs,Dst **MOVE** Rs,Dst	Move Word	0000010 | i | Rs | Dst
Modes: ☒ | ☒ | m | ☒ | ☒	Types: Rs: All*	Faults:

Dst ← Rs

All types (including CFUT) may be moved into memory.

READR Src,Rd **MOVE** Src,Rd	Read Register	000011 | Rd | 00 | Src
Modes: Register Mode	Types: Src: All but CFUT	Faults: CFUT

Rd ← Src

WRITER Rs,Dst **MOVE** Rs,Dst	Write Register	000100 | 00 | Rs | Dst
Modes: Register Mode	Types: Rs: All*	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Dst ← Rs

*Rs should have the proper type for the register described by Dst. Type checking is gone only for the address and IP registers, when the unchecked flag is off.

All types (including CFUT) may be moved to other locations.

RTAG Src,Rd	Read Tag	000101 | Rd | i | Src
Modes: R | A | m | i | c	Types: Src: All but CFUT	Faults: CFUT

Rd ← INT:tag(Src)

Note that access is allowed to the address register, immediate, and constant modes, but these operations are not very useful since address registers always have an ADDR tag, while immediates always have an INT tag and constants also have fixed tags.

WTAG Rs,Src,Rd	Write Tag	000110 | Rd | Rs | Src
Modes: R | A | m | i | c	Types: Rs: All; Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB RANGE

Rd ← Src:Rs

Src should be an integer between 0 and 15, inclusive. Src must be an integer unless the U flag is set. Rs can be any type.

LDIP Src	Load IP	000111 | i | 00 | Src
Modes: R | A | m | i | c	Types: Src: IP	Faults: CATASTROPHE TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

IP ← Src.

Src should be an IP-tagged value.

LDIPR Src	Load IP from Register	001000 | 00 | 00 | Src
Modes: Register Mode	Types: Src: IP	Faults: CATASTROPHE TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

IP ← Src.

Src should be an IP-tagged value.

CHECK Rs,Src,Rd	Check Tag	001001 | Rd | Rs | Src
Modes: R | A | m | i | c	Types: Rs: All; Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Rd ← BOOL:tag(Rs)=Src. Src must be an integer unless the U flag is set.

CARRY Rs,Src,Rd	Carry from Add	001010 | Rd | Rs | Src
ADD Rs,Src,Rd	Add	001011 | Rd | Rs | Src
SUB Rs,Src,Rd	Subtract	001100 | Rd | Rs | Src
Modes: R | A | m | i | c	Types: Rs: INT; Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB OVERFLOW

Add: Rd ← Rs+Src

Sub: Rd ← Rs-Src

An overflow occurs in checked mode when the signed result isn't the sum/difference of the signed parameters.

Carry returns 1 if adding the two numbers would generate an unsigned carry and 0 otherwise. It should not be used in checked mode, as it causes an overflow under the same conditions that *add* overflows. *Add* and *sub* produce results modulo 2^{32} in unchecked mode. In unchecked mode the type of Rd is the same as the type of Rs.

MULH	Rs,Src,Rd	Multiply High	001110	Rd	Rs	Src
MUL	Rs,Src,Rd	Multiply	001111	Rd	Rs	Src

Modes:

R	A	m	i	c

Types:

Rs	Src
INT	INT

Faults: TYPE
CFUT FUT TAG8
TAG9 TAGA TAGB
OVERFLOW

Mul: Rd ← Rs*Src

An overflow occurs in checked mode when the signed result isn't the product of the signed parameters.

MulH returns the high 32 bits of a 64-bit product. It should not be used in checked mode, as it causes an overflow under the same conditions as *mul* overflows (i.e. when the signed 32*32 product doesn't fit in 32 bits). In unchecked mode *Mul* returns the lower 32 bits of the 64-bit product, while *MulH* returns the upper 32 bits of that product. In unchecked mode the type of Rd is the same as the type of Rs.

ASH	Rs,Src,Rd	Arithmetic Shift	010000	Rd	Rs	Src
LSH	Rs,Src,Rd	Logical Shift	010001	Rd	Rs	Src

Modes:

R	A	m	i	c

Types:

Rs	Src
INT	INT

Faults: TYPE
CFUT FUT TAG8
TAG9 TAGA TAGB
OVERFLOW

Ash: Rd ← Rs<<Src

Lsh: Rd ← Rs<<Src

Src may be negative and may be very large. It is *not* treated modulo 32; instead, Rs is shifted by Src bits to the left or right if Src is negative, whatever Src happens to be. For example, if Src=-50, Rd is set to 0 by LSH and by ASH when Rs≥0 and to -1 by ASH when Rs<0. ASH treats Rs as a signed quantity, while LSH treats it as unsigned. An overflow occurs when Src>0 and significant bits are shifted from the number; bits shifted to the right from the number are ignored. In unchecked mode the type of Rd is the same as the type of Rs, and Src is treated as if it were a signed integer.

ROT Rs,Src,Rd	Rotate	010010 | Rd | Rs | Src
Modes: R A m i c	Types: Rs: INT; Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Rd ← Rs rotated left Src bits

This is a rotate instead of a shift, so bits shifted out of the left side of Rs are shifted back at the right side. Src is an integer treated modulo 32 (since a rotate of 32 bits is the identity transformation). In unchecked mode the type of Rd is the same as the type of Rs.

AND Rs,Src,Rd	AND	011000 | Rd | Rs | Src
OR Rs,Src,Rd	OR	011001 | Rd | Rs | Src
XOR Rs,Src,Rd	XOR	011010 | Rd | Rs | Src
Modes: R A m i c	Types: Src / Rs: INT / INT; BOOL / BOOL	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

And:	Rd ← Rs&Src
Or:	Rd ← Rs|Src
Xor:	Rd ← Rs^Src

The operations are bitwise in unchecked mode and in checked mode when performed on integers. A TYPE fault occurs in checked mode if Rs and Src have different types. The type of Rd is the same as the type of Rs.

FFB Src,Rd	Find First Bit	011011 | Rd | i | Src
Modes: R A m i c	Types: Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Rd ← FFB(Src)

Rd is loaded with an integer value between 0 and 31, inclusive. This indicates how many bits must be traversed, going from left to right starting from bit 30, in order to find the first bit not equal to the sign bit (bit 31). (for example, FFB($80000000)=0, FFB($E0000000)=2, and FFB($20000000)=1) This is useful for normalizing floating point values.

NOT	Src,Rd	NOT	011100	Rd	i	Src

Modes: R A m i c	Types: Src: INT, BOOL	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Rd ← ~Src

This is a bitwise operation on integers. In unchecked mode all 32 bits are complemented for all input types except booleans, in which only the least significant bit is complemented.

NEG	Src,Rd	Negate	011101	Rd	i	Src

Modes: R A m i c	Types: Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB OVERFLOW

Rd ← -Src

Note that this operation can overflow if Src=$80000000.

LT	Rs,Src,Rd	Less than	100000	Rd	Rs	Src
LE	Rs,Src,Rd	Less than or Equal	100001	Rd	Rs	Src
GE	Rs,Src,Rd	Greater than or Equal	100010	Rd	Rs	Src
GT	Rs,Src,Rd	Greater than	100011	Rd	Rs	Src

Modes: R A m i c	Types: Rs / Src: INT / INT, BOOL / BOOL	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Lt:	Rd ← BOOL:Rs $<$ Src
Le:	Rd ← BOOL:Rs $\leq$ Src
Ge:	Rd ← BOOL:Rs $\geq$ Src
Gt:	Rd ← BOOL:Rs $>$ Src

A TYPE fault occurs in checked mode if Rs and Src have different types. In unchecked mode these instructions ignore tags and compare only the data fields.

EQUAL	Rs,Src,Rd	Equal	100100	Rd	Rs	Src
NEQUAL	Rs,Src,Rd	Not Equal	100101	Rd	Rs	Src

Modes:

R	A	m	i	c

Types:

Rs	Src
INT	INT
BOOL	BOOL
SYM	SYM

Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Equal: $Rd \leftarrow \mathrm{BOOL}{:}Rs = Src$

NEqual: $Rd \leftarrow \mathrm{BOOL}{:}Rs \neq Src$

A TYPE fault occurs in checked mode if Rs and Src have different types. In unchecked mode these instructions ignore tags and compare only the data fields.

EQ	Rs,Src,Rd	Pointer Equal	100110	Rd	Rs	Src
NEQ	Rs,Src,Rd	Pointer not Equal	100111	Rd	Rs	Src

Modes:

R	A	m	i	c

Types:

Rs	Src
All but CFUT FUT	All but CFUT FUT

Faults: CFUT FUT

Eq: $Rd \leftarrow \mathrm{BOOL}{:}Rs = Src$

NEq: $Rd \leftarrow \mathrm{BOOL}{:}Rs \neq Src$

Both the data and tag have to match for Rs to be considered equal to Src (in either checked or unchecked mode; this is different from the behavior of *Equal* and *NEqual* in unchecked mode).

XLATE Rs,Dst,C	Associative Lookup	101000 | C | Rs | Dst
Modes: R | A | ☒ | ☒ | ☒	Types: Rs: All but CFUT	Faults: CFUT XLATE

Dst ← associative_lookup(Rs); fault XLATE if no entry in table was found or if the associated data value for Rs is NIL.

The constant field C provides a way for the XLATE exception code to know what circumstances surrounded the failed translation so it can behave appropriately.

When XLATE'ing into an address register the key being XLATE'd is written into the corresponding ID register.

ENTER Src,Rs	Associative Enter	101001 | 00 | Rs | Src
Modes: R | ☒ | ☒ | ☒ | ☒	Types: Rs: All but CFUT; Src: All but CFUT	Faults: CFUT

Enter Rs and Src into the associative table so that associative_lookup(Src)=Rs. That is, Src is the key and Rs is the data. The slot used is picked at random except when associative_lookup(Src) already existed, in which case the old value is overwritten.

INVAL	Invalidate	101010 | 00 | 00 | 0000000

Invalidate all relocatable address registers (ones with the R bit set) on both priority levels by copying the R bit into the I bit.

PROBE Rs,Dst	Probe Associative Cache	101101 | 00 | Rs | Dst
Modes: R | ☒ | ☒ | ☒ | ☒	Types: Src: All but CFUT	Faults: CFUT

Attempt to find Rs in the XLATE cache. If Rs is there, Dst ←`lookup(Rs)`, else Dst ←`NIL`.

SUSPEND	Suspend	110000 | 00 | 00 | 0000000
		Faults: EARLY

Cease processing of the method and dequeue the message unless the B flag was set. Set SP to 0. Fault EARLY if the entire message has not yet been read. See the **SUSPEND** section for more details.

CALL Src	System Call	110001 | i | 00 | Src
Modes: R | A | m | i | ☒	Types: Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB RANGE

Fault using the vector at Src+128. Src must be an integer unless the U flag is set.

SEND Src	Send	110100 | P | i | Src
SENDE Src	Send and End	110101 | P | i | Src
SEND2 Src,Rs	Send 2	110110 | P | Rs | Src
SEND2E Src,Rs	Send 2 and End	110111 | P | Rs | Src
Modes: R | A | m | i | c	Types: Rs: All but CFUT; Src: All but CFUT	Faults: CFUT SEND

Send one or two words onto the network. When two words are sent, the one from Src is sent before the word in Rs; hence, please note the unusual assembler syntax order of Src and Rs. *SENDE* and *SEND2E* indicate the end of the message to the network hardware after the words they send. *SEND* and *SEND2* set the I Flag, while *SENDE* and *SEND2E* clear the I Flag. The op2 field is used to encode which message priority to send the message on.

BR `Src`	Branch	111000 | i | 00 | Src
Modes: R ☒ ☒ i ☒	Types: Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Branch forward Src words from the next word (i.e., when Src=0, the branch is to the next word) and clear the IP phase bit. Src must be a signed integer in checked mode.

BNIL `Rs,Src`	Branch if NIL	111010 | i | Rs | Src
BNNIL `Rs,Src`	Branch if Non-NIL	111011 | i | Rs | Src
Modes: R ☒ ☒ i ☒	Types: Rs: All but CFUT FUT; Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

BNIL: If Rs=NIL (both tag and data equal to 0), branch forward Src words (see *BR*).

BNNIL: If Rs≠NIL (either tag or data not equal to 0), branch forward Src words (see *BR*).

Note that unlike the other conditional branches, Rs may be any type except CFUT or FUT without causing a fault in checked mode.

BF `Rs,Src`	Branch if False	111100 | i | Rs | Src
BT `Rs,Src`	Branch if True	111101 | i | Rs | Src
Modes: R ☒ ☒ i ☒	Types: Rs: BOOL; Src: INT	Faults: TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

BF: If Rs=FALSE (bit 0 of data = 0), branch forward Src words (see *BR*).

BT: If Rs=TRUE (bit 0 of data = 1), branch forward Src words (see *BR*).

Rs must be a boolean in checked mode. In checked mode the branches branch on the state of bit 0 of Rs.

BZ	`Rs,Src`	Branch if Zero	$\boxed{111110}\boxed{i}\boxed{Rs}\boxed{Src}$
BNZ	`Rs,Src`	Branch if Nonzero	$\boxed{111111}\boxed{i}\boxed{Rs}\boxed{Src}$
Modes: $\boxed{R}\boxed{\times}\boxed{\times}\boxed{i}\boxed{\times}$		**Types:** Rs: INT Src: INT	**Faults:** TYPE CFUT FUT TAG8 TAG9 TAGA TAGB

Rs	Src
INT	INT

BZ: If data part of Rs=0, branch forward Src words (see *BR*).

BNZ: If data part of Rs≠0, branch forward Src words (see *BR*).

Rs must be an integer in checked mode.